7a. Instruction Set Architecture –
- translation software
- floating point representation

EECS 370 – Introduction to Computer Organization – Winter 2018

Mark Brehob, Reetu Das, Harry Davis

EECS Department
University of Michigan in Ann Arbor, USA

Announcements

- HW 2 is due Tuesday January 30th by 11:55 pm

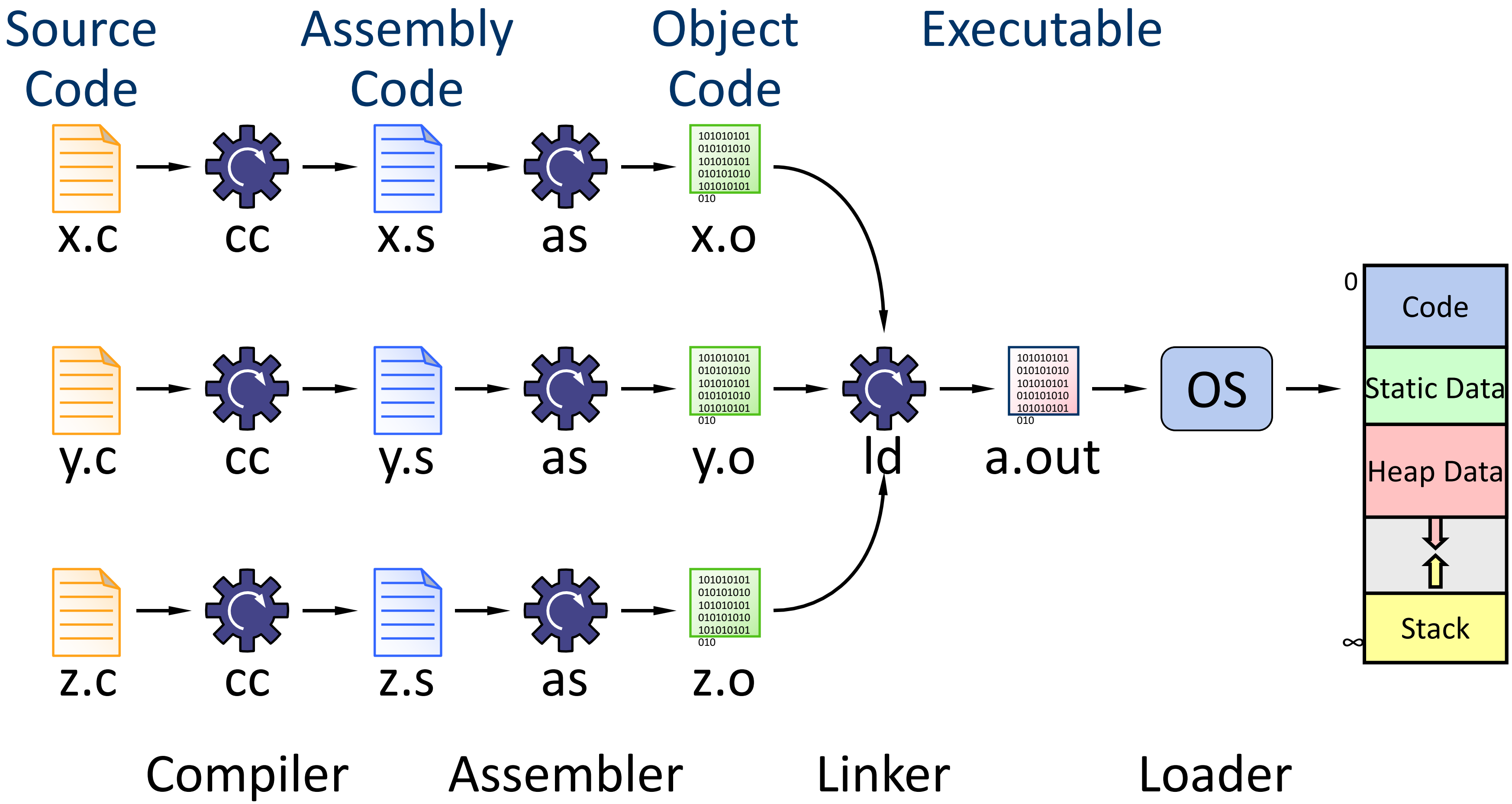
- Project 1m and 1s are due Thursday February 1st.

- Midterm Exam
  - 8:15pm (sharp) to 10:15pm on February 22nd
  - You have until January 30th to let us know of any conflict you may have.
    - Directions for doing so are on the website under “administrative requests”

Review

- How to support functions in assembly
  - Passing arguments and getting a return value
  - Using the stack and stack frame.
  - How to ensure that live values in registers are preserved after a function call.
- Introduction to linkers and loaders
  - Basic relationship of complier, assembler, linker and loader.
  - Object files
    - Symbol tables and relocation tables

Source to Process Translation



Linux (ELF—executable and linkable format) object file format

Object files contain more than just machine code instructions!

Header: (of an object file) contains sizes of other parts

Text: machine code

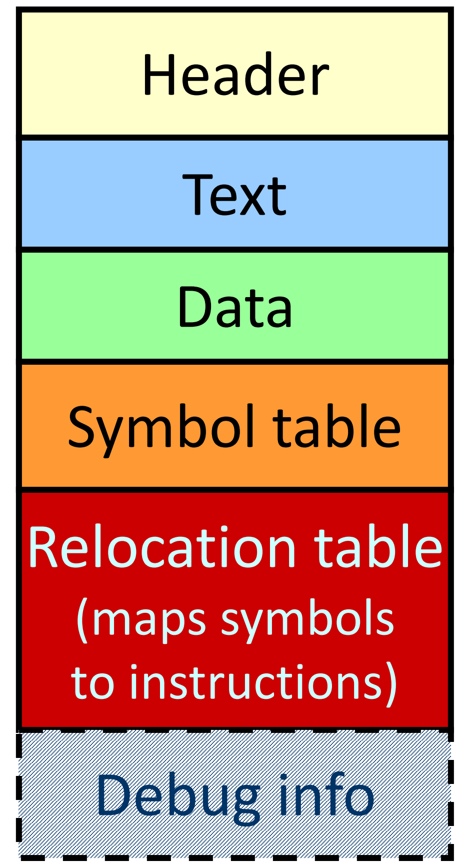
Data: global and static data

Symbol table: symbols and values

Relocation table: references to addresses that may change

Debug info: mapping of object back to source (only exists when debugging options are turned on)

Object code format



Assembly → Object file - example

Snippet of C

```
int X = 3;
main() {
  Y = X + 1;
  B();
  …
```

Snippet of assembly code

```
LDUR    X1, [X27, #0]
ADDI    X9, X1, #1
BL      B
```

Header	Name	foo	
	Text size	0x0C //probably bigger	
	Data size	0x04 //probably bigger	
	…		
Text	Address	Instruction	
	0	LDUR X1, [X27, #0] //X27 global reg	
	4	ADDI X9, X1, #1 //X9 local variable Y	
	8	BL B	
Data	0	X	3
Symbol table	Label	Address	Location
	X	0	Data
	B	-	-
	main	0	Text
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	X
	12	BL	B

Class Problem 1

- In the following files, which symbols will be put in the object file's symbol table? If its location is defined in this file, indicate if it would be in the data or text section.

Recall local variables are not in tables:
b in file 1 and
*e in file 2

file1.c
```
extern int bar(int);
extern char c[];
int a;
int foo (int x) {
    int b;
    a = c[3] + 1;
    bar(x);
    b = 27;
}
```

file 1 – symbol table

sym	**loc**
a	data
foo	text
c	-
bar	-

file2.c
```
extern int a;
char c[100];
void bar (int y) {
  char e[100];
  a = y;
  c[20] = e[7];
}
```

file 2 – symbol table

sym	**loc**
c	data
bar	text
a	-

Class Problem 2

Which lines / instructions are in the relocation table for each file?

file1.c

```
1 extern void bar(int);
2 extern char c[];
3 int a;
4 int foo (int x) {
5     int b;
6     a = c[3] + 1;
7     bar(x);
8     b = 27;
9 }
```

file 1 - relocation table

line	type	dep
6	ldur	c
6	stur	a
7	bl	bar

file2.c

```
1 extern int a;
2 char c[100];
3 void bar (int y) {
4   char e[100];
5   a = y;
6   c[20] = e[7];
7 }
```

file 2 - relocation table

line	type	dep
5	stur	a
6	stur	c

Note: in a real relocation table, the "line" would really be the address in "text" section of the instruction we need to update.

Some additional questions

```
file1.c
extern void bar(int);
extern char c[];
int a;
int foo (int x) {
   int b;
   a = c[3] + 1;
   bar(x);
   b = 27;
}
```

```
file2.c
extern int a;
char c[100];
void bar (int y) {
  char e[100];
  a = y;
  c[20] = e[7];
}
```

A) What if file2.c contains extern int j, but no reference to j?

A smart compiler would not put j in symbol table. A dumb one would. It's a benign issue, no harm done if j is in symbol table uselessly.

B) What if variable 'e' is static?

It is in the data section (no longer stack) and any reference to it is in relocation table. It is also in the symbol table so the address can be calculated, but it will be flagged not to be exported to other files, since the scope is local.

C) What if the externs in file1.c are deleted?

You should get a compile error

Loader

- Executable file is sitting on the disk
- Puts the executable file code image into memory and asks the operating system to schedule it as a new process
  - Creates new address space for program large enough to hold text and data segments, along with a stack segment
  - Copies instructions and data from executable file into the new address space (this may be anywhere in memory)
  - Initializes registers (PC and SP most important)
- Linking used to be straight forward, but times are changing; it is not simple anymore.
    - We now delay some of the linking to load time
    - Some systems even delay some code optimization (usually a compiler job) to load time
    - Loaders must deal with more sophisticated operating systems

Things to remember

- Compiler converts a single source code file into a single assembly language file
- Assembler handles directives (.fill), converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file
- Assembler does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses
- Linker enables separate compilation: Thus unchanged files, including libraries need not be recompiled.
- Linker resolves remaining addresses.
- Loader loads executable into memory and begins execution

Floating Point Arithmetic

Why floating point

- Have to represent real numbers somehow
- Rational numbers
  - Ok, but can be cumbersome to work with
  - Falls apart for sqrt(2) and other irrational numbers
- Fixed point
  - Do everything in thousandths (or millions, etc.)
  - Not always easy to pick the right units
  - Different scaling factors for different stages of computation
- Scientific notation: this is good!
  - Exponential notation allows HUGE dynamic range
  - Constant (approximately) relative precision across the whole range

Floating point before IEEE-754 standard

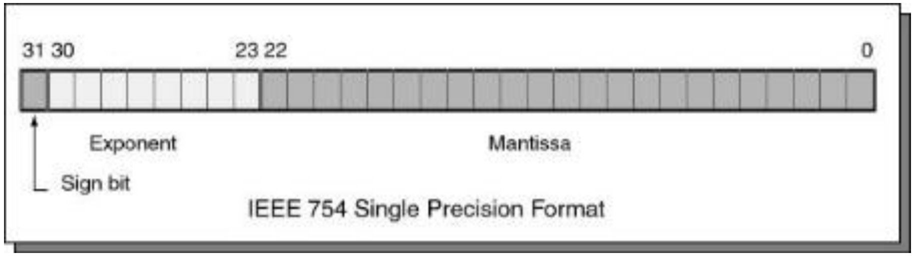
- Late 1970s formats
  - About two dozen different, incompatible floating point number formats
  - Precisions from about 4 to about 17 decimal digits
  - Ranges from about 10^{19} to 10^{322}
- Sloppy arithmetic
  - Last few bits were often wrong, and in different ways
  - Overflow sometimes detected, sometimes ignored
  - Arbitrary, almost random rounding modes
    - Truncate, round up, round to nearest
  - Addition and multiplication not necessarily commutative
    - Small differences due to roundoff errors

IEEE floating point

- Standard set by IEEE
  - Intel took the lead in 1976 for a good standard
  - First working implementation: Intel 8087 floating point coprocessor, 1980
  - Full formal adoption: 1985
  - Updated in 2008
- Rigorous specification for high accuracy computation
  - Made every bit count
  - Dependable accuracy even in the lowest bits
  - Predictable, reasonable behavior for exceptional conditions
    - (divide by zero, overflow, etc.)

IEEE Floating point format (single precision)

- Sign bit: (0 is positive, 1 is negative)
- Significand: (also called the *mantissa*; stores the 23 most significant bits after the decimal point)
- Exponent: used biased base 127 encoding
  - Add 127 to the value of the exponent to encode:
  - -127 $\rightarrow$ 00000000 1 $\rightarrow$ 10000000
  - -126 $\rightarrow$ 00000001 2 $\rightarrow$ 10000001
  - … …
  - 0 $\rightarrow$ 01111111 128 $\rightarrow$ 11111111
- How do you represent zero ? Special convention:
  - Exponent: -127 (all zeroes), Significand 0 (all zeroes), Sign + or -



Floating Point Representation

$10.625_{10} \Rightarrow 1010.101_{2}$

1.010101×2^{3}

This must be a 1!
So don't store it.

+/-	Exponent (3)	Significand (1010101)
1 bit	8 bits	23 bits

Floating Point Representation

$10.625_{10} \Rightarrow 1010.101_2$

1.010101×2^3

This must be a 1!
So don't store it.

+/-	Exponent (3)	Significand (1010101)
1 bit	8 bits	23 bits

$10.625_{10} = 0 \quad 10000010 \quad 01010100000000000000000$

Class Problem

- What is the value (in decimal) of the following IEEE 754 floating point encoded number?

1	**10000101**	**01011001000000000000000**

Floating point multiplication

- Add exponents (don't forget to account for the bias of 127)
- Multiply significands (don't forget the implicit **1** bits)
- Renormalize if necessary
- Compute sign bit (simple exclusive-or)

Floating point multiply

$$10.625_{10} = 1010.101_2$$

$$10_{10} = 1010_2$$

0	10000010	01010100000000000000000
	+	×
0	10000010	01000000000000000000000
	-127	
0	10000101	10101001000000000000000

```
   1 0 1 0 1 0 1
×          1 0 1
-----------------
   1 0 1 0 1 0 1
 1 0 1 0 1 0 1 0 0
-----------------
 1 1 0 1 0 1 0 0 1
```

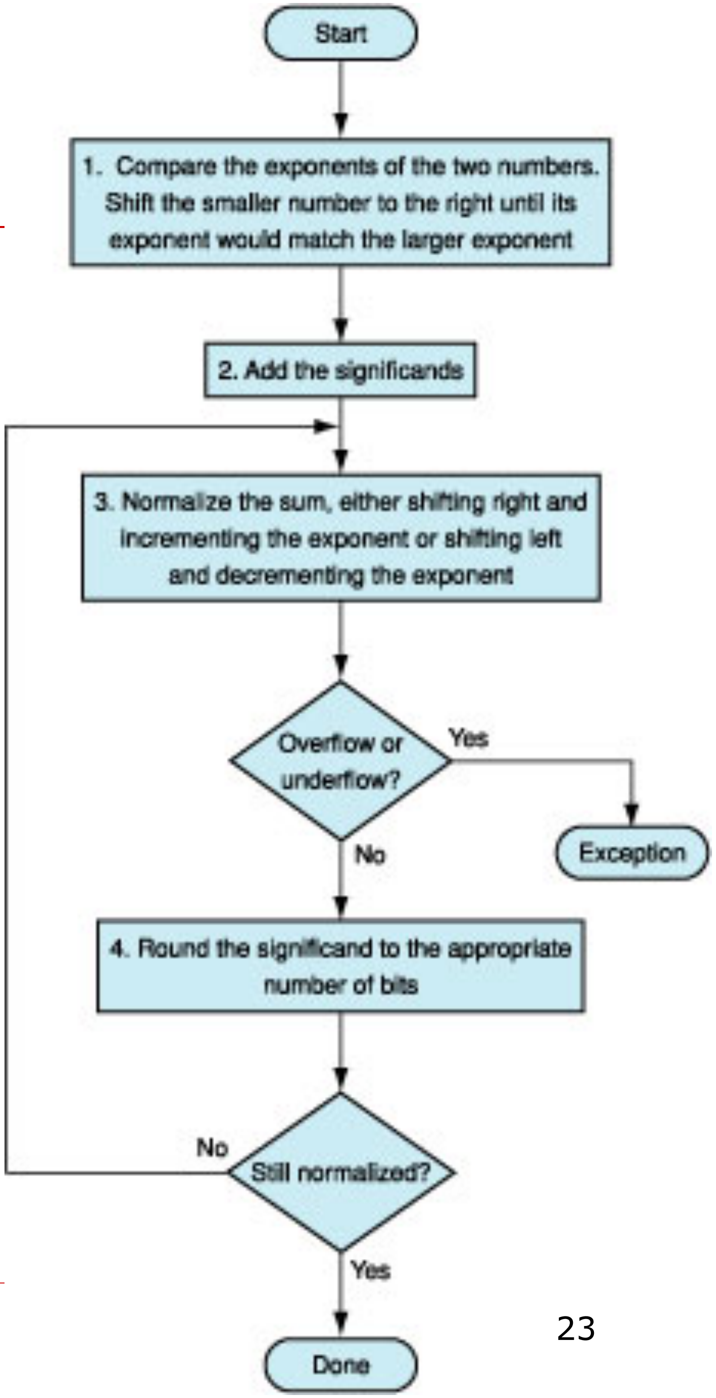
$$1101010.01_2 = 106.25_{10}$$

Floating point addition

- More complicated than floating point multiplication!
- If exponents are unequal, must shift the significand of the smaller number to the right to align the corresponding place values
- Once numbers are aligned, simple addition (could be subtraction, if one of the numbers is negative)
- Renormalize (which could be messy if the numbers had opposite signs; for example, consider addition of +1.5000 and – 1.4999)
- Added complication: rounding to the correct number of bits to store could denormalize the number, and require one more step

Floating point Addition

1. Shift smaller exponent right to match larger.
2. Add significands
3. Normalize and update exponent
4. Check for “out of range”



Class Problem

Show how to add the following 2 numbers using IEEE floating point addition: 101.125 + 13.75

Class Problem

101.125	0	10000101	10010100100000000000000
13.75	0	10000010	10111000000000000000000

Shift by 6-3 = 3

Shift mantissa by difference in exponent

00110111000000000000000

Note: When shifting to the right, the first shift should put the implicit 1, then 0's

Sum Significands

$$
\begin{array}{r}
1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1 \\
+\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0 \\
\hline
1\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1
\end{array}
$$

Sum didn't overflow, so no re-normalization needed

0	10000101	11001011100000000000000

= 114.875

Class Problem

Show how to add the following 2 numbers using IEEE floating point addition: 117.125 + 13.75

Class Problem

117.125	0	10000101	11010100100000000000000
13.75	0	10000010	10111000000000000000000

Shift by 6-3 = 3

Shift mantissa by difference in exponent

00110111000000000000000

Note: When shifting to the right, the first shift should put the implicit 1, then 0's

Sum Significands

```
  1 1 1 0 1 0 1 0 0 1
+ 0 0 0 1 1 0 1 1 1 0
---------------------
1 0 0 0 0 0 1 0 1 1 1
```

0	10000110	00000101110000000000000

= 130.875

Sum overflows, re-normalize by adding one to exponent and shifting mantissa by one

More precision and range

- We have described IEEE-754 binary32 floating point format, commonly known as "single precision" ("float" in C/C++)
  - 24 bits precision; equivalent to about 7 decimal digits
  - $3.4 * 10^{38}$ maximum value
  - Good enough for most but not all calculations
- IEEE-754 also defines a larger binary64 format, "double precision" ("double" in C/C++)
  - 53 bits precision, equivalent to about 16 decimal digits
  - $1.8 * 10^{308}$ maximum value
  - Most accurate physical values currently known only to about 47 bits precision, about 14 decimal digits

Extreme floating point

- binary128, "quad precision"; recent addition to standard:
  - 113 bits precision, about 34 decimal digits
  - $1.2*10^{4932}$ maximum value
  - Very rarely used, but some computations require extreme accuracy to limit cumulative roundoff error
- Another recent addition was binary16, "half precision"
  - 11 bits precision, about 3.3 decimal digits
  - 65504 maximum value
  - Used in graphics processors for accurate rendering of scenes with a large dynamic range in lighting levels.
  - Minimizes storage per pixel

Single ("float") percision